AD-A057 319 WHARTON SCHOOL PHILADELPHIA PA DEPT OF DECISION SCIENCES F/G 9/2
IMPLEMENTING ALERTING TECHNIQUES IN DATABASE SYSTEMS,(U)
FEB 77 O P BUNEMAN, H L MORGAN N00014-75-C-0462
UNCLASSIFIED 77-03-04 NL

| OF |
ADA
057319

END
DATE
FILMED
9-78
DDC

(12)

(6) IMPLEMENTING ALERTING TECHNIQUES IN DATABASE SYSTEMS,

(10) O. Peter/Buneman
Howard Lee/Morgan

(14) 77-03-04

78 06 21 011

Department of Decision Sciences
The Wharton School
University of Pennsylvania
Philadelphia, PA 19174

(12) 20 p.

December 1975
Revised June 1976
Revised (11) Feb 1977

Research supported in part by the Office of Naval Research under Contract (15) N00014-75-C-0462.

(This Paper supersedes 75-12-04) A037 437

408 757

Implementing Alerting Techniques in Database Systems

O. Peter Buneman and Howard Lee Morgan

1.0 INTRODUCTION

We usually regard databases and other information systems as passive in that they "speak only when spoken to." In the real world however, information is often spontaneously offered to us by friends and colleagues who, being aware of our interests, draw our attention to changes that may affect us. In a similar vein, this paper discusses the subject of "alerters", which can be used in a database management system to provide the ~~same~~ capability of informing a user when a specified state of the world (as reflected in the database) is reached. We feel that the ability to develop and implement alerters efficiently should prove a powerful adjunct to modern database technology. This paper describes ~~our~~ recent research in building database management systems which incorpotate alerting features, and provides a framework within which this and similar research efforts can be evaluated.

Some simple examples may help to clarify the concept. Consider the commands:

1. "Report the name and temperature of any station at which the temperature falls below 10 degrees Centigrade."

2. "Report the number and owner of any account from which more than $500 is withdrawn."

78 06 21 011

Such commands make most sense in the case of large shared databases, in which many users are entering updates at the same time that still other users may be querying the database. Rather than require a user repeatedly to enter a query of interest (e.g., 1 or 2 above), we propose that alerters be used. This changes the character of the DBMS from a *passive* one, where the DBMS only responds to queries, to an *active* one, in which the system may at any time send a message to alert the user to some condition of interest.

To our knowledge, there is no system which permits the construction of alerters in the sense in which we have described them: as dynamically defined user programs which monitor online databases. However, some of the ingredients are available in various programming language and database management systems. Several file management languages, ASAP[4], RPG[14] and some versions of COBOL[3] for example, permit the construction of programs which take some action, usually issuing a report, and which are activated whenever a record which satisfies specified conditions is read in during a sequential pass of the file. This technique is widely known as exception reporting and may be used for certain kinds of error handling. Hammer[7] has recently discussed the use of alerting type features for preventing semantic errors from creeping into databases.

The PL/I language has the ON condition, which permits the programmer to process interrupts together with SIGNAL, which lets the programmer generate interrupts. Common uses for the ON condition are for debugging, e.g. for reporting when the value of a variable exceeds some bound, for the trapping of end of file and similar I/O conditions, and for handling machine faults (e.g. arithmetic overflow, division by zero).

Both PLANNER and CONNIVER [15] allow the programmer to set up antecedent theorems, often called "demons", which are programs that monitor additions and deletions to an incore set of lists, and require certain actions to be taken before the addition or deletion can occur.

Finally, one of the proposed features of System R[1] is the TRIGGER command. This would permit one to put semantic integrity constraits of the type Hammer has discussed into a DBMS. As far as the authors could determine, this has not been implemented, and it is not expected to be in the early versions of this experimental system.

2.0 THE DEFINITION OF SIMPLE ALERTERS

The two examples in the introduction indicate that an alerter should be able to monitor <u>changes</u> to the database rather than its instantaneous state. We may therefore define an alerter as a condition, program pair where the condition may be any predicate involving two consecutive states of the database. If we refer to these states as OLD and NEW, the condition of the low-temperature alert may be more precisely defined as:

(OLD: temp > 10) and not(NEW: temp > 10)

Similarly it may be desirable for the program part, in this case "report the name and temperature", to access both old and new states of the database.

To allow arbitrary conditions and programs to reside in the data base may cause severe problems of efficiency and implementation. A user might create a condition requires that a scan of a substantial number of records in the data base be made on each update. Moreover a user defined program, if it is to reside entirely in the database, will require checks to be made that it is neither in an error state, nor consuming so much time that the performance of the database is degraded for other users.

We shall first describe how a certain class of alerters, _simple_ alerters can be implemented without the need for continuous polling of the database. We assume that the database consists of a number of record classes; each class possessing a number of fields. This is compatible with the relational model, though, as a matter of implementation, we are not restricting ourselves to relational systems. Formally, a simple alerter may be specified by the syntactic form of its condition. That condition may be constructed with boolean and arithmetic expressions involving (a) constants and (b) the field names of one record class prefixed by OLD: or NEW:. Thus simple alerters can monitor

1. one or more fields in an arbitrary record in that class, e.g. monitor the temperature at any station.

2. addition of a record to a class., e.g. monitor new stations added.

3. deletion of a record from a class

Examples of such conditionals in a field-monitoring alerter of type 1 above would be

```
NEW:temp - OLD:temp >20

(OLD:balance + OLD:limit > 0) and (NEW:balance + NEW:limit < 0)
```

This syntax is clumsy, and we shall shortly improve it; note however that it does not allow the construction of alerters which are triggered on aggregate properties of a record class e.g. average age > 60, nor does it permit monitoring of two or more classes simultaneously. The important advantage of this restriction is that only a small computational overhead is required on each update transaction in order to implement simple alerters.

2.1 Implementation Of Simple Alerters.

In this section we shall discuss the simplest possible extension of a database that allows for efficient alerting. To do this we must view the database and its users as disjoint sets of programs. Each user must be capable of efficiently sending and receiving messages from the database. Among the messages that a user program may send to the database will be:

U1. FIND(<class>,<record-specs>) to find the record in <class> conforming to <record-specs>.

U2. ADD(<class>,<record>) to add <record> to <class>.

U3. DELETE(<class>,<record-specs>) to delete a record.

U4. MODIFY(<class>,<record-specs>,<record>) to modify the record in <class> with <record-specs> to its new value, <record>.

and the database may send to the user:

D1. FOUND(<record>) in response to a FIND command.

D2. ERROR(<description>) in response to user errors where <description> may be 0 if no error has occurred.

We should emphasize that this is not meant to be an exhaustive list, nor is it meant to represent a query language; it is a description of a minimal class of messages that may usefully be passed between a database managment system and a user program. It should be apparent that the items in the first list correspond to subroutine calls, and the items in the second list correspond to subroutine returns in a more conventional database system in which the data manipulating subroutines are called directly by user programs.

The question now arises of how much of the computation required for the maintenance of an alerter should be performed by the database and how much should be performed by user programs. If we require that an alerter is entirely maintained by user programs then we are demanding that each user has a program which is continuously and independently monitoring the database, a solution which requires constant polling of the data and the transmission of a large number of the messages just described. The other extreme is to have the database maintain an arbitrary user program, and this may lead to the problems of implementation and efficiency described above.

As a first step in finding the right compromise, we could demand that a description of every update is sent out by the database to every user. A user defined alerter is then translated into a program which monitors these updates. This can be done because a simple alerter as defined above requires only information about the record involved in an update for checking its condition. Thus the low-temperature alerter could be maintained by a program which receives copies of the OLD and NEW versions of a record immediately before and after a MODIFY by 1. checking if the records are weather-station records 2. checking if the temperature field has changed and 3. applying the condition of the alert to the two temperature values.

As a more practical solution, we require that the first two of these checks should be performed by the database and that notices of updates are only sent out to users to whom the update is relevant. This will cut down both the amount of time wasted by user programs in checking irrelevant updates and the amount of message passing that needs to be done. For example, a user with no alerters will never receive an update notice.

Consider what this means for a simple modify alert: the list of messages that user programs may send to the database is extended by:

U5. MODALERT(<name>,<class>,<field-list>) meaning that the user program wishes to be notified of any updates which cause a field in <field-list> of the <class> records to change. The alerter is identified by <name>.

if the database detects such a change, it sends to that user a message of the form

U3. MODIFIED(<name>,<record1>,<record2>) which indicates that in response to the user's MODALERT request <name>, a relevant modification has occurred and that <record1> and <record2> are the two versions of the record before and after this change.

In order to complete the catalog of messages needed to implement simple alerting, we need also to add to the user message list ADDALERT(<name>,<class>) and DELETEALERT(<name>,<class>). To the database message list we add ADDED(<name>,<record>) and DELETED(<name>,<record>).

The crucial observation to make here is that we do not require to scan all the records in a given class upon each update. With a set of simple alerters the time overhead for each update to records in a given class is proportional only to the the number of alerters placed on that class. It is this observation which makes alerting practicable for very large databases. Once a sharable database with a message passing facility has been built, the implementation of this simplified (as far as the database is concerned) alerting system is not hard. We shall briefly describe one such implementation.

3.0 WAND

The Wharton Alerting Netwok Database (WAND) consists of an implementation of the CODASYL report by Gerritsen[5] with a simple alerting system which was added by Cortes. The whole system may be diagrammed as follows:

```
USER 1

USER 2
  .                ALERTING          DBTG ROUTINES
  .                 SYSTEM           AND DATABASE
  .
USER n
```

The alerting system may be regarded as monitoring transactions with the database, in particular it expands a MODIFY command into a more complicated set of DBTG instructions. To the alerting system, an alerter is defined by the following attributes:

1. USERID. The identification of the user issuing this alert.

2. NAME. The name that user has given the alerter.

3. TYPE. e.g. modify, add or delete.

4. CLASS. The class of records to be monitored.

5. FIELDS. A list of fields in that class to be monitored. This list is empty for add and delete alerters.

A modify alerter is indexed under the pair CLASS, FIELD so that there will be a list of alerters associated with each (<class>,<field>) pair. When a STORE command (this is the DBTG command which causes an update) is performed by some user, the following steps are taken:

The alerting system first does a FIND(<record>,<record-specs>) to get a copy of the record, OLDR say, which is to be modified.

NEWR is set to contain a copy of <record> and the MODIFY instruction is performed.

The appropriate ERROR code is returned to the user, if the modification was unsuccessful no further action is taken.

OLDR and NEWR are compared to determine which fields that have been modified. (For technical reasons, we cannot determine these fields by looking at the MODIFY command itself.)

The union of the lists of alerters indexed under (<class>,<field>) for all <field>s on this list is taken.

For each alerter on this union the message MODIFIED(NAME,OLDR,NEWR) is sent to the appropriate USERID.

In practice, this computation represents a small overhead on each update; the method in part resembles that recently proposed by Zobrist and Carlson [17] to detect the occurrence of certain configurations in a game of chess.

One of the advantages of Cortes's [5] implementation is that it uses the CODASYL database to store and retrieve the information required about alerters so that the alerting system itself requires no independent storage and requires only a small amount of code to implement this method of alerting. In an application in which the updates are frequent, it will be advantageous to keep those data structures associated with alerters in core as they are accessed on each update. In some versions of WAND this happens automatically because of an adaptive filing system [8] which has been added.

There is no reason why the same principles could not be used in conjunction with other database systems. The problem is that a general translator is required to transform a user-defined alerter into the appropriate user programs which are capable of responding for example to MODIFIED messages from the database.

We now describe another system which we have built with Stan Cohen and which is more sophisticated in its ability to translate such user-defined alerters into the appropriate internal code.

4.0 THE LDEMON SYSTEM

LDEMON is written in LISP and will accept such messages as:

```
ALERT "frostwarning" (TEMP < 10)
          (WRITE "Temp at" STATION "is now" TEMP)
```

The basis of LDEMON is a set of programs for constructing and manipulating records and manipulating messages which may be sent to it by other processes. Record classes may be dynamically defined, and the LISP functions for accessing their fields are automatically constructed. In addition, procedures are provided for adding, deleting and modifying records. This system is consistent with a relational model and a (not especially efficient) set of programs have been added to the system which provide for the usual relational constructs.

LDEMON allows a special class of programs, called <u>demons</u>. These may contain conditionals of the form (JCOND <condition> <actions>); meaning that <actions> are to be performed whenever <condition> has just become true and the form of <condition> is similar to that described in the preceding section. Changes which cause a particular state of the database to occur can therefore be monitored.

The distinction between alerters and demons lies in what kinds of action they are allowed to take. For a demon, the action part may be anything including a program which may itself modify the database. This is unacceptable for alerters since, as we have noted, an incorrect program, may "hang" the database indefinitely. The following describes what happens in LDEMON when an alerter is added to the system and subsequently activated.

4.1 Adding An Alerter To The LDEMON

1. On receiving an alerter message, the alerter is "stamped" with the identification of the user who sent it. It is then checked for consistency with the database schema. That is, field names must be known to the system, and related to the proper record types. Any inconsistency causes an error message to be returned.

2. The action portion is decoded, and user commands are checked. At this time, in the LISP based system, a demon is constructed which will actually send the appropriate response if the appropriate conditions are met.

3. The alerter is indexed under the (field, record class) pair for each field name which appears in the condition.

4. At this point, an "ALERT SET" message is sent back to the user.

4.2 Triggering An Alerter

1. When a modifying update is to be performed, a copy of the old values is made in a temporary area.

2. The update is completed and a copy of the updated values is also placed in temporary store.

3. By examination of the updating transaction, a list of affected (changed) fields is constructed.

4. Any alerter indexed under the appropriate record class and field name is activated; i.e., its condition is evaluated with respect to the records in temporary storage and, if satisfied, the function which sends the appropriate messages is invoked.

Note that we can no longer look upon the whole alerting process of

LDEMON as being synchronous: alerter messages are effectively issued after the update has taken place. Most importantly, we cannot use an alerter in LDEMON for preventing an update. Much of Morgan's work[9] was dedicated to resolving conflicts among the "event sequenced programs", which, in the present terminology would be the actions. Since alerters cannot interfere with updates, and can only have the "side effect" of sending a message, there is no conflict problem as far as the database system is concerned.

5.0 COMPLEX ALERTING.

There are many useful predicates which cannot be handled by simple alerters. In some cases the implementation of such alerters poses real problems of efficiency. We give here an informal classification of the types of predicate we would like to handle but which require a more global view of the database system and the transactions it supports.

1. Structural Alerters. These require that an alerter simultaneously monitor several record classes and require knowledge of the schema or structure of the database. E.g., a doctor gets a patient with chicken pox (either by the doctor acquiring the patient, or an existing patient acquiring chicken pox). In the context of a relational database Buneman and Clemons[2] have shown that there are a number of simple ways of reducing the amount of computation required to maintain a structural alerter.

2. Statistical alerters. E.g., let me know when the average temperature in the Northeast drops by *more than* 10 degrees. Clearly, one can come up with efficient methods for handling simple statistical functions (with averages by maintaining a special counter which is updated appropriately), but the more difficult the function, the less likely that such a routine exists, and the more likely that a scan of the database may be required.

3. Transaction spanning alerters. E.g., let me know when a bank balance drops by more than 10,000 in any 24 hour period. This requires a constant rolling history for 24 hours.

4. Pattern recognition. Monitoring for the occurence of specific patterns in the database. This is closest to CONNIVER's original intentions for using demons. Again, efficient implementation requires a lot of problem specific knowledge.

5. Time based alerting. This is similar to transaction spanning, in that one often does not require instant alerting, but rather alerting that doesn't come too late. For example, one may wish to be informed within one week if a particular check is cashed.

6.0 CONCLUSIONS

We have shown that a useful class of alerters (simple alerters) can be implemented, with a computing time overhead which is independent of the number of records in a large database. The extra time required to handle alerting for each update is dependent only on the number and complexity of the alerter conditions themselves.

An experimental system with these features has been implemented in LISP and running for over a year. A similar system, working with DBTG databases, has been completed. Both of these are in use and in use within a command and control testbed environment.

We feel that alerting features are a natural extension to database technology, and that they should be planned for in any major database implementations.

7.0 REFERENCES

1. Astrahan, M.M. et al. "System R: Relational Approach to Database Managment" ACM Transactions on Database systems. 1, 2 (June 1976).

2. Buneman, O.P, and E. K. Clemons, "Efficiently Monitoring Relational Databases," Decision Sciences working paper, University of Pennsylvania.

3. COBOL Language Reference Manual. IBM Corporation.

4. Conway, R., W. Maxwell and H. Morgan, "A Technique for File Surveillance," Proceedings of IFIP Congress 74, North-Holland Publishing Company.

5. Gerritsen, R., J. Ribiero and R. Cortes, WAND User's Guide, Working Paper No. 76-01-03, Department of Decision Sciences, The Wharton School.

6. Gerritsen, R. and H. Morgan, "Dynamic Restructuring of Databases using Generation Data Structures," Proceedings of ACM '76, ACM.

7. Hammer, M. "Error Detection in Database Systems," Proceedings of the 1976 National Computer Conference, AFIPS Press, (June 1976).

8. Lepoff, E. "The use of access frequencies in Hierarchical Memory Managment", Master's thesis, Moore School, University of Pennsylvania 1974

9. Morgan, H. "An Interrupt Based Organization for Management Information Systems," Comm. ACM 13, 12 (December 1970).

10. Morgan, H. L., "Event sequenced programming." Technical Report No. 119, Dept. of Operations Research, Cornell University, Ithaca, NY.

11. Morgan, H. L., "A generalized interrupt processor for Pl/I," Internal memorandum, Caltech, 1972.

12. Morgan, H. L. "DAISY: An Applications Perspective," Proceedings of the Wharton/ONR Conference on Decision Support Systems, (to appear 1976).

13. Morgan, H. L. and R. Wagner, "The Design of a High Performance Compiler for Pl/I," Proceedings of the 1971 Spring Joint Computer Conference, AFIPS Press.

14. RPG Language specification. IBM corporation C24 333706.

15. Sussman, G. and McDermott, D. "Why Conniving is better

than Planning," MIT Artificial Intelligence memo 255A (April 1972).

16. Zelkowitz, Marvin, "Reversible Execution as a Program Debugging Tool," Ph.D. Thesis, Cornell University, 1972.

17. A.L. Zobrist and Carlson, F.R., Detection of Combined Ocurrences, Comm. ACM. 20,1 (1977) 31-33